A Conservative Approach to Handle Full Functions in the Polyhedral Model

INRIA Research Report number 6814

Mohamed-Walid Benabderrahmane, Cédric Bastoul,
Louis-Noël Pouchet and Albert Cohen

ALCHEMY Group, INRIA Saclay Île-de-France and University of Paris-Sud 11,
{firstname.lastname}@inria.fr

Abstract. The Polyhedral Model is one of the most powerful framework for automatic optimization and parallelization of high-level programs. It is based on an algebraic representation of program parts and allows to achieve exact data dependence analysis and to apply complex sequences of optimizations seamlessly. After more than twenty years of research and development, this model is now quite mature and reaches production compilers as GCC 4.4 and its GRAPHITE framework. The main limitation of the Polyhedral Model is known to be its restricted application domain. Traditionally, it is used to manipulate very regular program parts only. The goal of this paper is to show this limitation is mostly artificial. We identify the main problem as the code generation step. We propose an extension to the polyhedral representation and to a code generation algorithm that allows to manipulate full functions in the Polyhedral Model.

Keywords Polyhedral model, code generation, irregular control flow.

1 Introduction

Complex program restructuring ability is needed for optimizing tools to cope with the growing complexity of modern architectures. The recent abundance of multicore processors in personal computers urges production compilers to provide such capability to efficiently exploit the computing potential of these architectures. It also urges software industry to provide efficient tools to help developers to manipulate and to optimize their programs.

Since the very first compilers, the internal representation of programs has been the Abstract Syntax Tree, or AST. In such representation, each statement appears only once even if it is executed many times (e.g., when it is enclosed inside a loop). *This is a limitation for program analysis.* For instance if a statement depends on another statement (i.e., they access the same memory location and at least one of these accesses is a write), we will consider both statements as single entities while the dependence relation may involve only few statement executions. *This is a limitation for program transformations.* Loop transformations operate on statement executions. For instance, because they consider all statement executions at once, several present day production compilers are not

able to achieve a basic loop fusion (that tries to merge the loop bodies of two loops) when the loop bounds of the two loops do not match. *This is a limitation for program manipulation flexibility.* Trees are very rigid data structures that are not easy to manipulate. Program transformation may require very complex transformations that will imply deep modifications of the control flow. Hence, for complex program restructuring, the need for a more precise, more flexible representation.

The Polyhedral Model is an alternative representation which combines analysis power, expressiveness and high flexibility. It is an algebraic representation of programs born in the late Sixties with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [20]. The Polyhedral Model is closer to the program execution itself by considering *statement instances*. Thanks to its strong mathematical foundations, it allows exact data dependence analysis and seamless application of complex sequences of optimizations that lead to many advances in automatic optimization and parallelization of programs [13, 6, 23, 17, 5, 24].

Most research works and existing frameworks based on the Polyhedral Model target code parts that exactly fit the model. Basically, only very regular codes (made of `for` loops with affine bounds and `if` conditionals with affine conditions) can be translated to a polyhedral representation. There are two main reasons to this situation: (1) a less strict program model would no longer allow an exact analysis and (2) there is no code generation scheme in the polyhedral model to generate dynamic control flows. We believe the main power of the Polyhedral Model is not to achieve exact data dependence analysis but to perform, as a single trivial step, sequences of complex optimizations. In this paper, we propose to relax the usual (restricted) application domain of the polyhedral model, by leveraging a conservative approach. We show how, thanks to slight changes to the representation itself and of a state-of-the-art code generation algorithm, full functions may benefit from the expressiveness and flexibility of the Polyhedral Model.

The paper is organized as follows. Section 2 introduces the classical polyhedral representation of programs and extensions to support irregular codes. Section 3 revisits the polyhedral framework to target full functions, from analysis to code generation. In section 4 we discuss the control overhead problem and solutions. Finally, section 5 discusses related work before concluding in section 7.

2 Polyhedral Representation of Programs

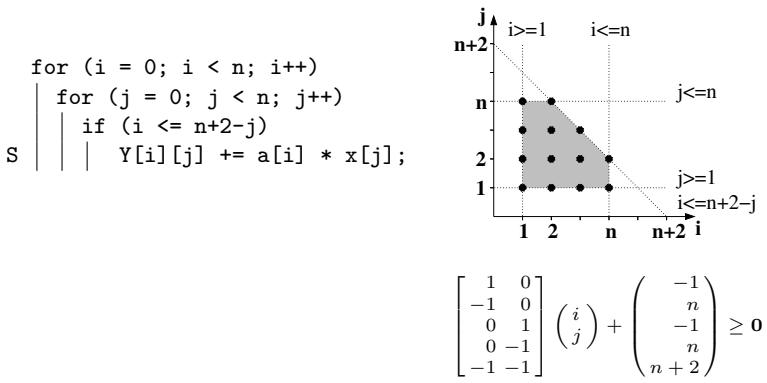
2.1 Static Control Parts

Static control parts (SCoP) are a subclass of general loops nests that can be represented in the Polyhedral Model. A SCoP is defined as a maximal set of consecutive statements, where loops bounds and conditionals are affine functions of surrounding iterators and the global parameters (constants whose values are unknown at compile time). The iteration domain of these loops can always be specified by a set of linear inequalities defining a polyhedron. The term polyhedron will be used to denote a set of points in a $\mathbb{Z}^n$ vector space bounded by affine

inequalities:

$$\mathcal{D} = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{Z}^n, A\boldsymbol{x} + \boldsymbol{a} \geq 0\}$$

where $\boldsymbol{x}$ is the iteration vector (the vector of the loop counter values), A is a constant matrix and $\boldsymbol{a}$ is a constant vector, possibly parametric. The iteration domain is a subset of the full possible iteration space: $\mathcal{D} \subseteq \mathbb{Z}^n$. Figure 1 illustrates the matching between surrounding control and polyhedral domain: the iteration domain in Figure 1(b) can be defined using affine inequalities that are extracted directly from the program in Figure 1(a). Each axis in the Figure 1(b) corresponds to a loop, and each dot is an iteration of the loop nest driving the execution of the statement S. The set of affine constraints may be represented with the matrix notation shown in Figure 1(b).



(a) Surrounding Control of S (b) Iteration Domain of S

Fig. 1. Static Control and Corresponding Iteration Domain

2.2 Relaxing the Constraints

The program model we target in this paper is general functions where the only control statements are `for` loops, `while` loops and `if` conditionals. This means no function calls are allowed and `goto`, `continue` and `break` statements have been removed thanks to some preprocessing. To move from static control parts to such general control flow we need to address two issues: (1) modeling loop structures with arbitrary bounds (typically `while` loops); and (2) modeling arbitrary conditionals (typically data-dependent ones). In both cases, it implies to not be anymore able to exactly characterize statically the iteration domain of statements, which remains the privilege of Static Control Parts.

First, we demonstrate that it is possible to express safe over-approximations of the iteration domains to allow the construction of a polyhedral representation in the case of arbitrary control-flows.

Modeling Arbitrary Loop Structure Any arbitrarily iterative structure such as `for` loops with non-affine bounds or `while` loops is actually amenable to polyhedral representation. As explained in Section 2.1 the iteration domain of a statement is a subset of $\mathbb{Z}^n$. The convex hull of all executed instances of any statement therefore remains a subset of $\mathbb{Z}^n$, so we can safely over-approximate the iteration domain of statements under a non-static loop as $\mathbb{Z}^n$. We actually choose to over-approximate it as $\mathbb{N}^n$ to match the standard loop normalization scheme, represented by the non-negative half-space polyhedron. Such over-estimate have been used in the same way by Griebl and Collard in the `while` loop parallelization context [18].

To guarantee that the program semantics will be preserved, we introduce an **exit predication** statement which bears the loop bound check. This statement is executed at the beginning of any iteration of the infinite loop, and exits the loop thanks to a `break` instruction if the loop conditional is no longer satisfied. This is summarized in Figure 2: we consider the original code in Figure 2(a) as the equivalent code in Figure 2(b) with the exit predicate `ep`. In the case of arbitrary `for` loops, initialization statements is inserted just before the loop and at the end of the loop body for the increment. Note that all statements in the body of the loop depends on the exit predication statement. The exit predicate is attached to the iteration domain of the predicated statements as illustrated in the example in Figure 2(c).

```
while (condition)
| S;
```

(a) Original Code

```
for (i=0;; i++) {
| ep = condition;
| if (ep)
| | S;
| else
| | break;
}
```

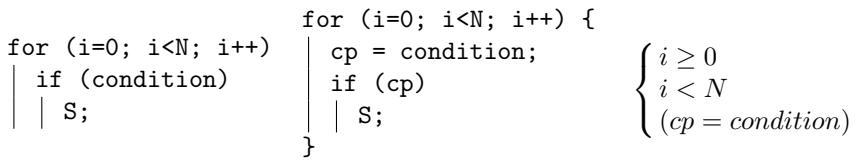
(b) Equivalent Code

$$\begin{cases} i \geq 0 \\ (ep = condition) \end{cases}$$

(c) Iteration Domain of S

Fig. 2. Exit-predication of a Statement Surrounded By a `while` Loop

Modeling Arbitrary Conditionals We apply a similar reasoning to represent non-affine conditionals. To model such a conditionally executed statement in the polyhedral representation we do not take into account the restriction imposed by the conditional to its iteration domain. Hence again, the iteration domain will be over-approximated, and we need to ensure the semantics is preserved. To do so we introduce a **control predication** which consists in predicating individually each statement dominated by the non-static conditional by its condition. This is summarized in Figure 3: we consider the original code in Figure 3(a) as the equivalent code in Figure 3(b) with the control predicate `cp`. The control predicate is attached to the iteration domain of the predicated statements as illustrated in the example in Figure 3(c).



(a) Original Code (b) Equivalent Code (c) Iteration Domain of S

Fig. 3. Control-predication of a Statement Surrounded By a Conditional

These examples highlight the fact that limitation to purely static control-flow remains artificial: at the expense of over-approximations of the iteration domain, *any kind of non-recursive control flow can be manipulated using a polyhedral representation*.

Being able to safely describe a polyhedral representation of the convex hull of the dynamic control flow is only the first step towards supporting full functions into the polyhedral framework. In the following Section we present the necessary and sufficient modifications of the framework that allow to *optimize* general codes with polyhedral techniques. Our goal is to show that, provided a conservative analysis, only the code generation step needs to be altered to enable the application of any polyhedral optimization technique on full functions.

3 Revisiting the Polyhedral Framework

The Polyhedral Model is a three steps framework. First, the Program Analysis part aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some optimizing or parallelizing algorithm use the analysis to restructure the programs in the Polyhedral Model during a Program Transformation Step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program. Targeting full functions requires to revisit the whole framework, from analysis to code generation.

3.1 Program Analysis

Once a function has been translated to the Polyhedral Model with the control and exit predicate extensions as described in Section 2.2, data dependence analysis must be performed.

One of the benefit of limiting to SCoPs is the easiness of computing data dependence. Array access functions are forced to be affine, hence it is possible to compute, *at the finest granularity*, on which executed instance of a given statement any other instance depend [11, 12].

As we broaden the set of handled programs, we have to deal with dynamic behavior (e.g., `while` loops) and structural complexity (e.g. subscript of subscript, as in `A[B[i]]`). As a result, an exact analysis is no more possible statically. Instead, we rely on a *conservative* policy, over-estimating data dependences and subsequently forbidding some transformations when semantics safety is unsure.

Conservative policies are widely used in compilation to achieve an approximate analysis of programs without slowing down the compiler. GCD-test [2] or I-test [22] are popular examples of such analysis for array references: they can state thanks to a fast GCD computation that two references do not depend on each other, then safely consider a dependence relation exists otherwise. When dedicated preprocessing techniques fail to simplify complex array references (typically subscript of subscript or linearized subscripts) it is usual to consider the reference as an access to a single variable, i.e., to suppose that the whole array is read or written. In the same way, when *array recovery* fails to translate pointer-based accesses to explicit array references [14], it is usual to consider a dependence between the pointer access and every previously accessed references. Eventually, it is possible to handle any kind of data access in a conservative way.

A conservative approach for irregular control flow analysis is to consider that additional statement instances are executed. As long as the additional statements do not modify directly the control flow (as `break`, `continue` or `goto` statements), their only side effect is to add data dependences to the analysis. Therefore in such situation it is safe to consider irregular conditions (from `while` loops as well as `if` conditionals) are always true.

In this paper, we translate the program control structures in such a way we only have to deal with regular `for` loops, regular `if` conditionals and infinite `for` loops. Irregularity has been spread thanks to control and exit predicates to the iteration domains of irregular-control-surrounded statements. One can achieve a naive but simple conservative analysis by considering an altered representation of the input irregular program called *abstract program*. We build this representation from the original program in this way:

1. Introduce control and exit predicates as described in Section 2.2.
2. Predicate evaluations are considered as statements that writes the predicate.
3. Irregular data accesses are transformed conservatively (e.g., array with complex accesses are considered as single variable accesses).
4. Predicated statements are considered to read their predicates.

Writing and reading predicates ensure the semantics is preserved when a statement modifies an element necessary for the predicate evaluation. Lastly we perform on this representation classic data dependence elimination techniques like array privatization [1] then exact data dependence analysis [12].

We illustrate the construction of the abstract program for conservative data dependence analysis in Figure 4. The considered program in Figure 4(a) is an optimized version of the Outer Product Kernel in the case one vector contains some zeros. The conditional introduces irregular control flow that usually prevents considering such kernel in the Polyhedral Model. The first step is to introduce a control predicate and to attach it to the predicated statements. The predicate evaluation is considered as a statement as shown in Figure 4(b). Lastly, we consider the value of the predicate is read by each predicated statement and that the predicate is always true for conservative data dependence analysis as shown in Figure 4(c). Figure 4(c) presents the information sent to the data dependence algorithm (everything is regular): for each statement, its iteration domain and the

sets of written and read references. We may use well known techniques to remove some dependences. In this example we can privatize `p` to remove loop-carried dependence and parallelize the code or even interchange the loops.

```
for (i = 0; i < N; i++)
  if (x[i] == 0)
    for (j=0; j < M; j++)
      A[i][j] = 0;
  else
    for (j=0; j < M; j++)
      A[i][j] = x[i] * y[j];
```

(a) Outer Product Kernel

```
for (i = 0; i < N; i++)
  p = (x[i] == 0);
  for (j=0; j < M; j++)
    if (p)
      A[i][j] = 0;
  for (j=0; j < M; j++)
    if (!p)
      A[i][j] = x[i] * y[j];
```

(b) Using a Control Predicate

```
for (i = 0; i < N; i++)
  S0: Written = {p}, Read = {x[i]}
  for (j=0; j < M; j++)
    S1: Written = {A[i][j]}, Read = {p}
  for (j=0; j < M; j++)
    S2: Written = {A[i][j]}, Read = {x[i],y[j],p}
```

(c) Abstract Program For Conservative Data Dependence Analysis

Fig. 4. Abstract Program Construction For Irregular Outer Product Analysis

Discussion Many previous works aim at providing less naive and conservative solutions to avoid, as much as possible, to consider additional dependences. Griebl and Collard proposed a solution in the context of `while` loops parallelization, focusing on control flow [18]. Collard et al. extended this approach to support complex data references [9]. Other techniques aim at removing some dependences as, e.g., Value-based Array Data Dependence Analysis [25], Array Region Analysis [10] Array SSA [21] or Maximal Static Expansion [3]. These techniques would expose their full potential in the context of manipulating full functions in the Polyhedral Model to minimize the unavoidable conservative aspects. Furthermore, combining these static analyses with dynamic dependence tests [28, 27] into hybrid polyhedral/dynamic analyses remains to be investigated.

3.2 Program Transformation

A (sequence of) program transformation(s) in the polyhedral model is represented thanks to a set of affine functions, one for each statement, called scheduling, allocation, chunking, etc. depending on the technique. In this paper we will use the generic term *scattering functions*. Scattering functions depend on the surrounding loop counters of their corresponding statement and map each run-time statement instance to a logical execution date. The literature is full of algorithms

to find such functions dedicated to parallelization, data locality or global performance improvement [13, 23, 17, 5, 24]. The conservative approach allows to reuse most existing techniques based on the Polyhedral Model and multi-dimensional scattering **directly**.

However, managing `while` loops, that are translated into unbounded `for` loops requires a slight adaptation to preserve the expressiveness of any form of affine scattering. This is particularly highlighted in the context of one-dimensional affine functions. Using one-dimensional scattering, it is necessary to know the upper bounds of the loops to be able to reorder them. For instance let us consider the pseudo-code in Figure 5(a) composed of two loops enclosing two statements, `S1` and `S2`. To implement a transformation such that the loop enclosing `S2` will be executed before the loop enclosing `S1`, we need the logical dates of the instances of `S1` to be *higher* than those of the instances of `S2`. Such transformation may be implemented by the scattering functions $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$. In these functions, the i part ensures the instances of a given statement are executed in the same order as in the original code, and the upper bound $Up2$ of the second loop is used to ensure the loop of `S1` *starts* after the end of the loop of `S2`. The target code is shown in Figure 5(b), where the variable t represents the logical time.

```
for (i = 0; i < Up1; i++)
  S1;
for (i = 0; i < Up2; i++)
  S2;
```

(a) Original Program

```
for (t = 0; t < Up2; t++)
  i = t;
  S2;
for (t = Up2; i < Up2 + Up1; i++)
  i = t - Up2;
  S1;
```

(b) Loop Reordering with scattering $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$

Fig. 5. Loop Reordering Using One-Dimensional Scattering

In this work, we may consider `for` loops with no upper bounds. It is not possible in this way to reorder those loops respectively to other loops (bounded or unbounded) using one-dimensional schedules[1]. To overcome this issue, we propose to consider a virtual parametric upper bound `w`, the same for all unbounded `for` loops with the constraint that `w` is strictly greater than all upper bounds of bounded `for` loops. The `w`-parameter will be considered during the program transformation and code generation steps. It will be removed during a dedicated stage of code generation as detailed in Section 3.3. This parameter has to be chosen strictly greater than other loop bounds to ensure a fusion between a bounded and an unbounded loop will always be partial (hence the code generation step will

[1] It is easy to remove the limitation using more dimensions, but several algorithms to compute scattering functions are based on one-dimensional scattering only, and some others rely on the full expressiveness of single dimensions.

always be able to re-create the unbounded part). A single `w`-parameter for multiple unbounded loops is enough to be able to reorder them relatively to each other by using coefficients of this parameter (e.g., to reorder three unbounded loops, we can use scattering functions like $\theta_{S1}(i) = i$, $\theta_{S2}(i) = i + w$ and $\theta_{S3}(i) = i + 2w$). The `w`-parameter allows to reuse any of the existing algorithms supporting parameters to compute scattering functions in our irregular, conservative context.

3.3 Code Generation

After the conversion of a program to the polyhedral model, the application of a transformation on it leads to a new coordinate system for the points of the original iteration domains [4]. Code generation consists then in producing a program which scans these transformed iteration domains. It amounts to finding a set of nested loops visiting each integral point for each polyhedron once and only once. This is a critical step in the polyhedral framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion typically because a large code may pollute the instruction cache.

Among existing methods to scan polyhedra and generate code, the extended Quilleré et al. algorithm is considered now as the most efficient algorithm for regular programs [26, 4]. This algorithm is not able in its original form to generate semantically correct code for our extended polyhedral representation, as special care is needed to handle properly predicates and their impact on the generated control-flow. Nevertheless, it is possible to extend this algorithm to scan and generate regular codes corresponding to the over-estimates of the iteration domains. A post-processing of the output of the Quilleré et al. algorithm completes the process and guarantee semantically correct code generation.

We first provide a short description of the Quilleré et al. algorithm, then we present a new extension of this algorithm to generate irregular programs.

Quilleré et al. Algorithm Quilleré et al. proposed the first code generation algorithm to build the target code directly free of redundant control, in contrast of other approaches starting from a naive code and trying to improve it [26]. The main part of the algorithm is a recursive generation of the scanning code, maintaining a list of polyhedra from the outermost to the innermost loops. It makes a strong use of polyhedral operations that can be achieved by, e.g., PolyLib[2] [29]. Figure 6 provides a short description of the Quilleré et al. algorithm.

An Irregular Extension of the Quilleré et al. Algorithm Previous approaches to model irregular codes employed complex representations that did not allow any easy modification of the extended Quilleré algorithm to generate the code, therefore failing at generating good quality code. By relaxing the static constraints thanks to simple predication, we make possible and *even natural* the

[2] PolyLib is available at http://icps.u-strasbg.fr/PolyLib

CodeGeneration: Build a polyhedra scanning code AST without redundant control.

Input: a polyhedron list, a context C, the current dimension d.
Output: The AST of the code scanning the polyhedra inside the input list.

1. Intersect each polyhedron in the list with the context C;
2. Project the polyhedra onto the outermost d dimensions;
3. Separate these projections into disjoint polyhedra (this generates loops for dimension d and new lists for dimension $d+1$);
4. Sort the loops so that their textual order respects the lexicographic order;
5. Recursively generate loop nests that scan each new list with dimension $d+1$;
6. Return the AST.

Fig. 6. Quilleré et al. Algorithm

adjustment of the code generation algorithm. This adaptation takes into account the additional data dependences on control predicates. The price to pay is displacing the problem of modeling data dependent non-affine conditions into legality constraints. The two tasks to achieve are the semantically-correct generation of control predicates and exit predicates, the latter allowing to reconstruct while loops in the generated code.

Generation of Arbitrary Conditionals Generating arbitrary conditionals is silently handled: the control predicate is available as a statement information, and the only task is to generate the `if` instruction containing the predicate around the concerned statement.

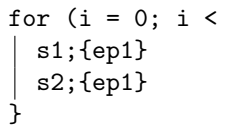
Generation of `while` Loop Structure The task of generating `while` loops starts by identifying irregular over-approximated code. This identification can be done by searching for loops with the `w` parameter, introduced in Section 3.2 as an upper bound of `for` loops, since this latter form is a result of the transformation of `while` loops to fit in the model done in the previous steps of the polyhedral framework. In a second step, we have to identify exit predicates corresponding to each `while` loop. This information is easily extracted because attached to the iteration domain of each statement which is part of a `while` loop in the original program.

However, due to the separation step of the extended Quilleré et al. algorithm, several statements with different exit predicates could be found in the same iteration domain without corresponding to the same `while` loop. So we need to separate these statements and generate the appropriate `while` loops. we distinguish three main cases of separation that involve exit predicates:

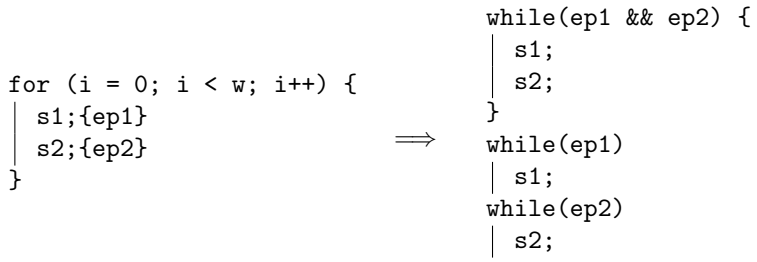
1. If all statements of the loop have the same exit predicate, no case distinction is needed during the separation phase. The predicate is therefore considered as the exit predicate of the generated `while` loop. Figure 7(a) is an example of such a case.
2. If statements or block of statements have different exit predicates, this means (1) they belong to different `while` loops; and (2) these statements can be

executed in any order since if they have different exit predicates and the same iteration domain, they should have the same scattering function as Quilleré et al. algorithm takes account only iteration domains and scattering functions when processing.
For this second case, we can proceed to a separation quite similar to separation of polyhedra in a regular case. More exactly, it consists in scanning the domain where both predicates are true at the same time, thanks to the intersection of two polyhedra, i.e., the space of common points. Then, we scan domains where only one of the two predicates is true, thanks to the differences between polyhedra. Figure 7(b) shows separation of while loops based on exit predicates attached to statements `s1` and `s2`.

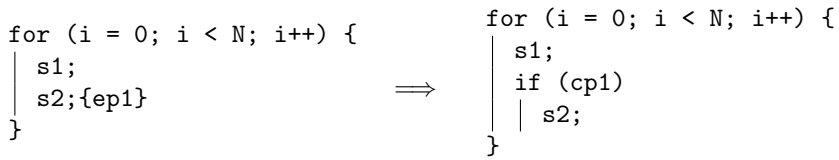
3. If some statements have exit predicates while some others do not have any, this means a regular `for` loop has been fused with a part of a `while` loop. In such a case, we find a statement with an exit predicate attached to it without identifying the `while` loop (by identifying the `w` parameter). The exit predicate is transformed here into a control predicate. Figure 7(c) illustrate this case.



(a) Same Exit Predicates



(b) Different Exit Predicates



(c) An Exit Predicate Inside A Regular Loop

Fig. 7. Separation of `while` Loops

Re-injecting irregular control inside the generated code is likely to bring high control overhead as it is inserted close to the statement, at the innermost level.

Discussion The semantics of transformations involving `while` loops is particular: fusion of such loops should be performed only if the loops can be executed in any order (in Figure 7(b), the order of the last two `while` loops is arbitrary). Also, when the transformation states the loop may be run in parallel (e.g., no scattering functions means all loops are parallel) it means that, except what is necessary for the predicate evaluation, iterations of the loop may be run in parallel (this allows basic parallelization, e.g., a process devoted to the predicate computation that spread bundles of full iterations to different processors).

4 Reducing Control Overhead

The underlying principle of converting programs to the extended polyhedral representation is to conditionally execute statements depending on the value of a given predicate, which is not necessarily statically computable. To put the program into the model, we extensively predicate statements regardless of the control overhead we introduce. We rely on post-pass optimizations to harness that overhead, in order to generate efficient code.

We discuss two main optimizations, namely the *computation of the predicate value* and the *placement of control predicates*. A preliminary for those optimizations to be performed is the gathering of the set of *read* and *written* variables, for each statement *and each predicate*. It means we have to analyze the statement content to extract the variables involved. Obviously, the optimality of our optimization processes is constrained by the accuracy of this analysis. For instance we do not deal yet with control overhead optimizations of codes containing non-strictly aliased pointers, as the problem of pointer aliasing makes very difficult to compute correctly the set of read and written variables.

4.1 Computing the Value of Predicates

One origin of computation overhead induced by predicating a statement is the re-computation of the value of the predicate p if its value has not been modified (and therefore it will obviously be evaluated to the same value). To address this problem we decouple the *computation* of the predicate value from its evaluation. We first define the set of variables used to compute the predicate value. Let p be a predicate used to guard a statement, $\mathcal{V}_p$ is the set of variables used to compute p. For instance, for the predicate `p = x + 2 * y + b[i]` (where `i` is the *generated* iterator name), $\mathcal{V}_p = \{x, y, b, i\}$.

The algorithm operates on the generated abstract syntax tree (AST), in a two-step process. The first step consists of introducing a statement in the AST to compute the value of p, for each predicated statement. We guarantee optimality by ensuring that it is not possible to execute `p` less times while still preserving the program semantics. This is done by putting the statement `p` at the highest tree level such that no statement dominated by `p` modifies any of the variables in $\mathcal{V}_p$. The second step consists in eliminating duplicated predicate computations when a given predicate is used from multiple calling sites. We proceed by inspecting the AST for all `p` statements (involving the same predicate p), and checking if any of the variables in $\mathcal{V}_p$ is ever assigned in any execution path between two occurrences of `p`. If not, then the second occurrence can be safely removed.

As a result of this optimization, the computation of the value of each predicate is optimally minimized (given the accuracy of $\mathcal{V}_p$ computation). The check of the predicate value before each executed instance of a predicated statement is reduced to a simple test instruction over a scalar.

4.2 Predicate Placement

The second critical optimization is to reduce the number of executed checks on the value of a predicate. To do so, we hoist the conditional `if (p)` to the highest possible level in the AST, provided $\mathcal{V}_p$. A typical example is the case of all reachable instances of a given loop being predicated by the same p, which is never modified during the loop execution. The instruction `if (p)` can then be hoisted outside the loop, dramatically reducing the control overhead. We proceed by merging under a common conditional all consecutive statements (under the same loop) which involve the same predicate, such that none of the statements modify the predicate value. If all statements inside a loop are under the same conditional and this conditional does not depend on neither the loop iterator nor any of the statements under it, then the conditional can be safely moved around the loop instead. This optimization is reminiscent of classical if-hoisting compiler techniques, and it is efficiently performed as a code generation optimization pass. We extended the code generation tool CLooG [4] to support these extensions.

5 Related Work

Many works aims at optimizing irregular codes, but only few of them are based on the Polyhedral Model. Most irregular polyhedral techniques were developed in the context of `while` loop parallelization. Collard explored a speculative approach to parallelize loops nests with `while` loops [8, 7]. The idea is to allow a speculative execution of iterations which are not in the iteration domain of the original program. This method leads to more potential parallelism than with traditional polyhedral methods, at the expense of an invalid space-time mapping that are fixed thanks to a backtracking policy.

In contrast to the speculative approach, Griebl et al. explore a conservative one. They try to enumerate a super set of the target execution space, and proposed solutions (1) to prevent iterations that are not in the target execution space, and (2) to take care of the termination of the target loops. For the first problem, they define what they call *execution determination* where they introduce a predicate to determine if a point in the iteration space can be executed or not. For the second point, they define and compute *termination detection*. Griebl and Lengauer [19] propose a solution using a communication scheme in a distributed memory model to determine the upper bounds of the target loops, but this solution increases the execution time of the scanning. Griebl and Collard [18] describe for the same problem, a scheme called counter scheme used for shared memory models. Griebl et al. [15, 16] present another one called maximum scheme which is a modification of counter scheme.

Both existing speculative or conservative methods and the conservative approach described in this paper are static techniques which analyse code at compile

time. Previous methods focus on the parallelization of arbitrary loop structure, with dedicated methods. In constrast in this paper we revisit the complete polyhedral framework to allow the manipulation of a wider class of programs than usual and to apply already existing optimization and parallelization techniques based on the Polyhedral Model.

6 Ongoing Work

The extension to the Polyhedral Model we present in this paper allows to deal with bigger SCoPs than usually because the SCoP limits are no more driven by control flow irregularities but by the length of the function. Hence the need for new optimizing techniques to deal with such large SCoPs. We propose methods to analyse and modularize the code as, thanks to our extended representation, codes are likely to become too large to be handled by existing high complexity techniques, e.g. for automatic parallelization. The main idea is to cut these large codes (as smartly as possible) into smaller parts or modules so we can optimize each part alone, and then globally considering all the parts together.

Next, instead of considering a purely conservative approach based on existing data dependence analysis techniques, we extend it to take advantage of the predicates. We are exploring two complementary approaches: a speculative approach (based on a static analysis helped by profiling) and a dynamic analysis approach (driven by the run-time behavior).

Lastly, on a more technical point of view, we are extending Clan[3], our loop analyzer tool, to extract easily the extended polyhedral representation from the dynamic control parts of high level programs.

7 Conclusion

A convenient representation is a must when one needs to achieve complex processing on some data. In signal processing, we use the Fourier Transform and its inverse to switch from time to frequency and back from frequency to time to work on the much more convenient frequency space. It has been extensively shown how arbitrarily complex sequences of optimizations are trivial to apply in the Polyhedral Model. Switching from source code or abstract syntax tree to the Polyhedral Model is the Fourier Transform of restructuring compilation. In this paper, we showed how full functions can be manipulated in the Polyhedral Model thanks to slight and natural extension to the classical representation.

We showed that representing programs in the polyhedral model is a matter of dependence analysis. Previous works did not deliver a complete polyhedral framework for irregular programs as they misled the control-flow management from the data-flow one. We introduced robust over-approximations of dynamic programs that allow to manipulate them within the polyhedral framework. We circumvented the burden of irregularity of the controls thanks to well-chosen predication, while postponing to data dependence analysis the task of efficently model arbitrary data-flows. We proposed a code generation scheme that supports those extensions while limiting the control overhead they may introduce.

[3] `http://www.lri.fr/ bastoul`

References

1. J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
2. U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.
3. D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *POPL'98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–106, San Diego, California, 1998.
4. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, september 2004.
5. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113. ACM, June 2008.
6. P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 1998.
7. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *In Proc. of the 1994 Scalable High Performance Computing Conf.* IEEE, 1994.
8. J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Program.*, 23(2):191–219, 1995.
9. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *PPOPP'95 Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 92–101, Santa Barbara, California, 1995.
10. B. Creusillet and F. Irigoin. Exact versus approximate array region analyses, lncs 1239. In *LCPC'96 9th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 86–100, 1996.
11. P. Feautrier. Dataflow analysis of scalar and array references. *Intl. Journal of Parallel Programming*, 20(1):23–53, february 1991.
12. P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *Intl. J. of Parallel Programming*, 21(5):313–348, oct 1992.
13. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, dec 1992.
14. B. Franke and M. O'Boyle. Array recovery and high level transformations for dsp applications. In *CPC'10 Intl. Workshop on Compilers for Parallel Computers*, pages 29–38, Amsterdam, January 2003.
15. M. Geigl, M. Griebl, and C. Lengauer. A scheme for detecting the termination of a parallel loop nest. In *Proc. GI/ITG FG PARS'98*, 1998.
16. M. Geigl, M. Griebl, and C. Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Comput.*, 25(12):1489–1510, 1999.
17. M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für mathematik und informatik, universität Passau, 2004.
18. M. Griebl and J. francois Collard. Generation of synchronous code for automatic parallelization of while loops. In *EURO-PAR '95, Lecture Notes in Computer Science 966*, pages 315–326. Springer-Verlag, 1995.
19. M. Griebl and C. Lengauer. On scanning space-time mapped while loops. In *CONPAR 94 - VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing*, pages 677–688, London, UK, 1994.

20. R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, july 1967.
21. K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. In *POPL'98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, San Diego, California, 1998.
22. X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 Intl. Conf. on Parallel Processing*, pages 204–211, St. Charles, august 1990.
23. A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, Stanford University, 2001.
24. L.-N. Pouchet, C. Bastoul, A. Cohen, and S. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008.
25. W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC'93 Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, LNCS 768*, pages 546–566, 1993.
26. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. Journal of Parallel Programming*, 28(5):469–498, october 2000.
27. L. Rauchwerger and D. A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI'95 Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, California, June 1995.
28. S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *Intl. J. of Parallel Programming*, 31(4), 2003.
29. D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.